1. Consider the following directed graph (digraph) \( G = (V, E) \):

![Graph Image]

a) What is the set of vertices? \( V = \)

b) An edge can be represented by a tuple (from vertex, to vertex [, weight]). What is the set of edges? \( E = \)

c) A path is a sequence of vertices that are connected by edges. In the graph \( G \) above, list two different paths from \( v_0 \) to \( v_3 \).

d) A cycle in a directed graph is a path that starts and ends at the same vertex. Find a cycle in the above graph.

2. Like most data structures, a graph can be represented using an array, or as a linked list of nodes. The array representation is a two-dimensional array (called an adjacency matrix) whose elements contain information about the edges and the vertices corresponding to the indices. (Python could use a list-of-lists)

a) Complete the following adjacency matrix for the above graph. (Here a missing edge is represented by \( \infty \).)

<table>
<thead>
<tr>
<th></th>
<th>( v_0 )</th>
<th>( v_1 )</th>
<th>( v_2 )</th>
<th>( v_3 )</th>
<th>( v_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_0 )</td>
<td>0</td>
<td>1</td>
<td>( \infty )</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>( v_1 )</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( v_3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( v_4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) What is the big-oh to determine the edge-weight between any two vertices?

c) What is the big-oh amount of storage used to store the adjacency matrix?

d) The linked representation maintains a linked-list (or Python dictionary) of vertices with each vertex maintaining a linked list of other vertices that it connects to. Complete the adjacency list representation below:

![Adjacency List Image]

e) What is the big-oh to determine the edge-weight between any two vertices in an adjacency list?

f) What is the big-oh amount of storage used to store the adjacency list?
3. Below is the textbook’s edge, vertex, and graph implementations.
a) How does this graph implementation maintain its set of vertices?

b) How does this graph implementation maintain its set of edges?

c) What is the big-oh to determine the edge-weight between any two vertices

---

```python
""" File: vertex.py """
class Vertex:
    def __init__(self, key, color = 'white',
                 dist = 0, pred = None):
        self.id = key
        self.connectedTo = {}
        self.color = color
        self predecessor = pred
        self.distance = dist
        self discovery = 0
        self finish = 0
    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight
    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])
    def getConnections(self):
        return self.connectedTo.keys()
    def getId(self):
        return self.id
    def getWeight(self,nbr):
        return self.connectedTo[nbr]
    def getColor(self):
        return self.color
    def setColor(self, newColor):
        self.color = newColor
    def getPred(self):
        return self.predecessor
    def setPred(self, newPred):
        self.predecessor = newPred
    def getDiscovery(self):
        return self.discovery
    def setDiscovery(self, newDiscovery):
        self.discovery = newDiscovery
    def getFinish(self):
        return self.finish
    def setFinish(self, newFinish):
        self.finish = newFinish
    def getDistance(self):
        return self.distance
    def setDistance(self, newDistance):
        self.distance = newDistance

""" File: graph.py """
from vertex import Vertex
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0
    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex
    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None
    def __contains__(self,n):
        return n in self.vertList
    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)
    def getVertices(self):
        return self.vertList.keys()
    def __iter__(self):
        return iter(self.vertList.values())
```

---
4. Graphs can be used to solve many problems by modeling the problem as a graph and using "known" graph algorithm(s). For example, consider the word-ladder puzzle where you transform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → PALL → PALE → SALE → SAGE.

We can use a graph algorithm to solve this problem by constructing a graph such that
- a word represents a vertex
- an edge represents?
- a word ladder transformation from one word to another represents?

4. For the words listed below, draw the graph of question 3

foul  fool  foil  fail  fall  pall  pope
cool  pool  poll  pale  sale  page  sage

a) List a different transformation from FOOL to SAGE

b) If we wanted to find the shortest transformation from FOOL to SAGE, what does that represent in the graph?

c) There are two general approaches for traversing a graph from some starting vertex $s$:
- Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from $s$, before finding all vertices a distance 2 from $s$, etc.
- Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a “dead end,” we backtrack to the deepest vertex that allows us to try a different path.

Which of these traversals would be helpful for finding the shortest solution to the word-ladder puzzle?